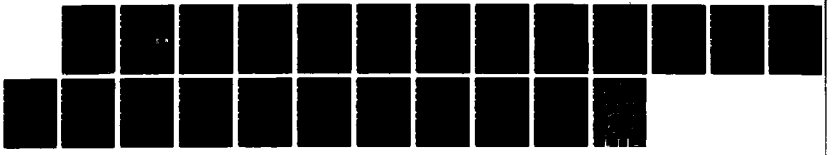
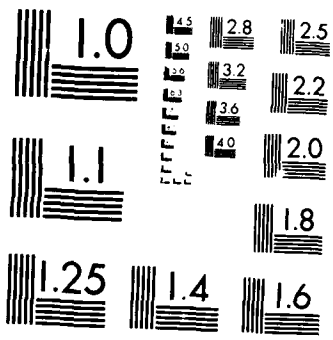
AD-A196 998 ABSTRACTION IN NUMERICAL METHODS(U) MASSACHUSETTS INST 1/1
OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB
M HALFANT ET AL OCT 87 AI-M-997 N00014-86-K-0180
UNCLASSIFIED F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(4)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 997	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Abstraction in Numerical Methods		5. TYPE OF REPORT & PERIOD COVERED: Memo
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Matthew Halfant and Gerald Jay Sussman		8. CONTRACT OR GRANT NUMBER(s): N00014-86-K-0180
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE: October, 1987 13. NUMBER OF PAGES: 18
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, VA 22217		15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Scheme, Abstraction, Programming Methodology, Richardson Extrapolation, LISP.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We illustrate how the liberal use of high-order procedural abstractions and infinite streams helps us to express some of the vocabulary and methods of numerical analysis. We develop a software toolbox encapsulating the technique of Richardson extrapolation, and we apply these tools to the problems of numerical integration and differentiation. By separating the idea of Richardson extrapolation from its use in particular circumstances we indicate how numerical programs can be written that exhibit the structure of the ideas from which they are formed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 997 *October 1987*

Abstraction in Numerical Methods

Matthew Halfant and Gerald Jay Sussman

Abstract

We illustrate how the liberal use of high-order procedural abstractions and infinite streams helps us to express some of the vocabulary and methods of numerical analysis. We develop a software toolbox encapsulating the technique of Richardson extrapolation, and we apply these tools to the problems of numerical integration and differentiation. By separating the idea of Richardson extrapolation from its use in particular circumstances we indicate how numerical programs can be written that exhibit the structure of the ideas from which they are formed.

Keywords: Scheme, Abstraction, Programming Methodology, Richardson Extrapolation, LISP.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.

Abstraction in Numerical Methods

Matthew Halfant and Gerald Jay Sussman

Artificial Intelligence Laboratory

and

Department of Electrical Engineering and Computer Science

Massachusetts Institute of Technology

October 1987

Abstract

We illustrate how the liberal use of high-order procedural abstractions and infinite streams helps us to express some of the vocabulary and methods of numerical analysis. We develop a software toolbox encapsulating the technique of Richardson extrapolation, and we apply these tools to the problems of numerical integration and differentiation. By separating the idea of Richardson extrapolation from its use in particular circumstances we indicate how numerical programs can be written that exhibit the structure of the ideas from which they are formed.

A numerical analyst uses powerful ideas such as Richardson extrapolation for organizing programs, but numerical programs rarely exhibit the structure implied by the abstractions used in their design. It is traditional practice in the domain of numerical methods for each program to be hand crafted, in detail, for the particular application, rather than to be constructed by mixing and matching from a set of interchangable parts. Such numerical programs are often difficult to write and even more difficult to read.

In this paper we illustrate how the liberal use of high-order procedural abstractions and infinite streams helps us to express some of the vocabulary and methods of numerical analysis. We develop a software toolbox encapsulating the technique of Richardson extrapolation, and we apply these tools to the problems of numerical integration and differentiation. By separating the idea of Richardson extrapolation from its use in particular circumstances we indicate how numerical programs can be written that exhibit the structure of the ideas from which they are formed.

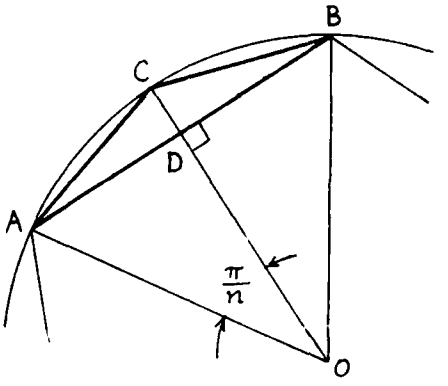


Figure 1: $|AB| = S_n$, $|AC| = |CB| = S_{2n}$

A first example: Archimedean computation of π

We begin with a playful example: approximating the value of π by the method of Archimedes. Let S_n be the length of one side of a regular n-sided polygon inscribed in a unit circle. As n approaches infinity, the semi-perimeter $P_n = nS_n/2$ approaches π. Applying the Pythagorean Theorem to right triangles ACD and ADO from figure 1 we derive the relation:

$$S_{2n} = \sqrt{2 - \sqrt{4 - S_n^2}}.$$

Equivalently,

$$S_{2n} = \frac{S_n}{\sqrt{2 + \sqrt{4 - S_n^2}}},$$

the latter form being preferred because it avoids the subtraction of nearly equal quantities as $S_n \to 0$. In Scheme [4] (the dialect of Lisp we use) we can write the transformation from S_n to S_{2n} as:

```
(define (refine-by-doubling s)                    ; s is a side
  (/ s (sqrt (+ 2 (sqrt (- 4 (* s s)))))))
```

Starting with a square (S_4 is simply $\sqrt{2}$), we want to form the sequence of side lengths $S_4, S_8, S_{16}, \cdots$. Such sequences are naturally represented with *streams*—effectively infinite lists whose terms are evaluated only on demand

(for a discussion of streams see [1]). We use a stream generator to produce the orbit of a starting value under repeated application of a transformation next:

```
(define (stream-of-iterates next value)
  (cons-stream value
               (stream-of-iterates next (next value))))
```

Now we can define the stream of side lengths:

```
(define side-lengths
        (stream-of-iterates refine-by-doubling (sqrt 2)))
```

and the corresponding stream of numbers of sides:

```
(define side-numbers
        (stream-of-iterates (lambda (n) (* 2 n)) 4))
```

Combining these termwise using `map-streams` lets us form the sequence of semi-perimeters $P_4, P_8, P_{16}, \cdots$, whose limit is π:

```
(define (semi-perimeter length-of-side number-of-sides)
  (* (/ number-of-sides 2) length-of-side))

(define archimedean-pi-sequence
  (map-streams semi-perimeter side-lengths side-numbers))
```

We can look at the results:

```
(print-stream archimedean-pi-sequence)

    ==>     2.82842712474619
            3.06146745892072
            3.12144515225805
            3.13654849054594
            3.14033115695475
            3.14127725093277    ; term 6
            ...
            3.14159265358862    ; term 20
            3.14159265358950
            3.14159265358972
            3.14159265358977
```

```
3.14159265358979
3.14159265358979
3.14159265358979
...
```

As expected, the sequence converges to π, but it takes 24 terms to reach the full machine precision. Imagine poor Archimedes doing the arithmetic by hand: square roots without even the benefit of our place value system! He would be interested in knowing that full precision can be reached on the fifth term, by forming linear combinations of the early terms that allow the limit to be seized by extrapolation.

To understand how this is done, rewrite the expression for the semi-perimeter P_n by using the Taylor series for $S_n = 2\sin(\pi/n)$ (see figure 1):

$$\begin{aligned} P_n &= (n/2)S_n \\ &= (n/2)(2\sin(\pi/n)) \\ &= \pi + A/n^2 + B/n^4 + \cdots \end{aligned}$$

where A and B are constants whose values aren't important here. As n gets larger, it is the A/n^2 term that dominates the *truncation error*—the difference between π and P_n for finite n. Whenever we double n, this principal component of the truncation error is reduced by a factor of 4; that knowledge allows us to combine successive terms of the sequence $\cdots P_n, P_{2n}, \cdots$ to eliminate entirely the effect of this quadratic term.

Specifically: multiply P_{2n} by 4, so that its $1/n^2$ term matches that of P_n; then subtract P_n to eliminate this term; after some slight rearrangement we get

$$\frac{4P_{2n} - P_n}{3} = \pi - \frac{B}{4n^4} + \cdots$$

The accelerated sequence P'_n is defined by the expression on the left; it approximates π with a truncation error that goes to 0 like $1/n^4$, which is much faster than before.

We could revise our program to compute the sequence $P'_4, P'_8, P'_{16}, \cdots$, and demonstrate that its convergence is more rapid than that of $P_4, P_8, P_{16}, \cdots$. However, this acceleration scheme is quite general, and we prefer to develop it away from the present context. Afterwards, we can apply it to this and other pursuits.

The method of Richardson extrapolation

Instead of phrasing our argument in terms of a parameter n that gets larger through successive doubling, we'll follow the convention of using a positive quantity h that approaches 0 through successive divisions by 2. In the example above, we need only make the identification $h = 1/n$. Sometimes n, sometimes h, will be the more natural, and we can formulate the problem however we prefer. Typically h will arise as a step size.

Now imagine that we seek the limit, as $h \to 0$, of some function $R(h)$, and that we pursue it by constructing the sequence $R(h), R(h/2), R(h/4), \cdots$. We suppose that each expression $R(h)$ represents the limiting value A with a truncation error that is analytic at $h = 0$:

$$R(h) = A + Bh^{p_1} + Ch^{p_2} + Dh^{p_3} + \cdots$$

The exponents $p_1, p_2, p_3, \cdots$ may well be just the natural numbers $1, 2, 3, \cdots$; but they might represent a subsequence—for example, the even numbers, as they did in the Archimedean example above. We assume this sequence is known. On the other hand, we need not know in advance the values of $A, B, C, \cdots$; indeed, our whole purpose is to determine A.

Knowing the truncation error as a power series in h allows us to eliminate the effect of the dominant term: we do this by subtracting the appropriate multiple of $R(h/2)$ from $R(h)$:

$$R(h) - 2^{p_1} R(h/2) = (1 - 2^{p_1})A + C_1 h^{p_2} + D_1 h^{p_3} + \cdots$$

or

$$\begin{aligned} R'(h) &= \frac{2^{p_1} R(h/2) - R(h)}{2^{p_1} - 1} \\ &= A + C_2 h^{p_2} + D_2 h^{p_3} + \cdots \end{aligned}$$

We view this as a transformation applied to (adjacent terms of) the sequence

$$R(h), R(h/2), R(h/4), \cdots$$

to produce a new sequence

$$R'(h), R'(h/2), R'(h/4), \cdots$$

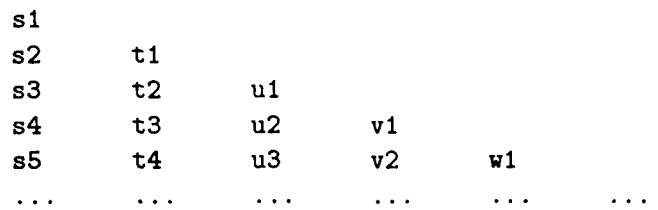


Figure 2: The Richardson Tableau

whose truncation error is now dominated by the term containing h^{p_2}. That is, the R' sequence converges faster than the original.

Now we come to Richardson's great idea: since R' has a truncation error dominated by $C_2 h^{p_2}$, we can apply the same idea again:

$$R''(h) = \frac{2^{p_2} R'(h/2) - R'(h)}{2^{p_2} - 1} = A + D_3 h^{p_3} + \cdots$$

yielding a sequence $R''(h), R''(h/2), R''(h/4), \cdots$ whose truncation error now has the dominant term $D_3 h^{p_3}$. And so on. Given the sequence $p_1, p_2, p_3, \cdots$, one can form a tableau (see figure 2) in which the original sequence appears as the vertical `s` column at the left; to the right is the derived `t` column; the `u` column is derived from `t` as `t` is from `s`, and so on. All columns converge to the same limit, $R(0+)$, but each converges faster than its predecessor. Thus, `s` converges with an error term $O(h^{p_1})$—that is, of *order* h^{p_1}; `t` converges with error term $O(h^{p_2})$; `u` has error term $O(h^{p_3})$, and so on.

By the way, if the sequence $p_1, p_2, p_3, \cdots$ is *not* known in advance, one can take the conservative approach of assuming it to be the natural number sequence $1, 2, 3, \cdots$; this leads at worst to the inefficiency of creating adjacent columns with the same error order. Alternatively, the appropriate exponent p for a given column can be inferred numerically from the early terms of that column; we have done that in our library, but do not pursue it here.

The Richardson Toolbox

Having sketched the basic ideas, we develop Richardson extrapolation as a set of tools that can be applied in diverse contexts. First, if we don't already have the sequence $R(h), R(h/2), R(h/4), \cdots$ we need to be able to make it,

given R and h. The name make-zeno-sequence derives from the suggestive connection to successive halving in the statement of Zeno's paradox.

```
(define (make-zeno-sequence R h)
  (cons-stream (R h) (make-zeno-sequence R (/ h 2))))
```

The basic operation of accelerating the sequence requires that we know the order of the dominant error term. Accelerate-zeno-sequence takes this order as an explicit argument.

```
(define (accelerate-zeno-sequence seq p)
  (let* ((2^p (expt 2 p)) (2^p-1 (- 2^p 1)))
    (map-streams (lambda (Rh Rh/2)
                   (/ (- (* 2^p Rh/2) Rh)
                      2^p-1))
                 seq
                 (tail seq))))
```

We can get our hands on the full tableau. In this case we iterate the application of accelerate-zeno-sequence to make an infinite sequence of accelerated sequences. In forming the full tableau, we make the simplifying assumption that the exponent sequence is arithmetic:

$$\{p_1,\ p_2,\ p_3, \cdots\} = \{p,\ p+q,\ p+2q,\ p+3q, \cdots\}$$

Hence it can be specified by the initial order p and an increment q. In typical cases, q is 1 or 2.

The procedure make-zeno-tableau accepts as arguments the original Zeno sequence along with the characterizing order p and increment q; it returns the sequence of accelerated sequences.

```
(define (make-zeno-tableau seq p q)
  (define (sequences seq order)
    (cons-stream seq
                 (sequences (accelerate-zeno-sequence seq order)
                            (+ order q))))
  (sequences seq p))
```

Finally, the procedure `first-terms-of-zeno-tableau` produces a sequence `s1, t1, u1, v1, w1, ...` of the first terms taken from each of the accelerated sequences. The sequence of first terms converges at a remarkable rate: not only are the first n terms of the original truncation-error series removed, but the remainder is effectively divided by $2^{n^2/2}$ (see appendix for details).

```
(define (first-terms-of-zeno-tableau tableau)
  (map-streams head tableau))

(define (richardson-sequence seq p q)
  (first-terms-of-zeno-tableau (make-zeno-tableau seq p q)))
```

Archimedes revisited

Before proceding any further, let's see how well this does on our original example. We apply `richardson-sequence` to the `archimedean-pi-sequence` previously computed and examine the result:

```
(print-stream (richardson-sequence archimedean-pi-sequence 2 2))

    ==>    2.82842712474619
           3.13914757031223
           3.14159039312994
           3.14159265328605
           3.14159265358979
           3.14159265358979
           3.14159265358979
           ...
```

As indicated earlier, full precision is reached on the 5th term (although we need to compute the 6th to know that we've reached it, and may want to compute the 7th just to be sure). What remains now is to establish some algorithmic means for assertaining when a limit has been reached.

Completing and applying the Richardson toolbox

Now that we have the idea under control, we must fill in our Richardson toolbox to allow its application in a variety of situations. We need ways of

extracting our best estimate of the limit from a sequence. One simple criterion that may be used is this: We declare convergence when two consecutive terms are sufficiently close.

Alas, the notion of sufficient closeness is slightly sticky: relative accuracy is what's generally wanted, but that fails in the case of a sequence with limit 0. One way around the difficulty is to use the metric

$$\frac{|h_1 - h_2|}{(|h_1| + |h_2|)/2 + 1}$$

to measure the distance between two numbers h_1 and h_2:

```
(define (close-enuf? h1 h2 tolerance)
  (<= (abs (- h1 h2))
      (* .5
         tolerance
         (+ (abs h1) (abs h2) 2))))
```

This criterion amounts to relative closeness when the numbers to be compared are large, but makes a graceful transition to absolute closeness when the numbers are much smaller (in magnitude) than 1.

Using this or any similar predicate, we construct our limit detector:

```
(define (stream-limit s tolerance)
  (let loop ((s s))
    (let* ((h1 (head s)) (t (tail s)) (h2 (head t)))
      (if (close-enuf? h1 h2 tolerance)
          h2
          (loop t)))))
```

A more cautious version of the limit detector would require close agreement for three or more successive terms (we've been bitten ourselves by accidental equality of the first two terms of a sequence). Actually, there's another modification we'll be forced to make very shortly: we'll need an optional final argument `m` that forbids `stream-limit` from examining more than the first `m` terms of the sequence before returning an answer.

Given `stream-limit`, the following combination proves useful for finding the Richardson limit of a function. The arguments `ord` and `inc` are our previous p and q.

```
(define (richardson-limit f start-h ord inc tolerance)
  (stream-limit
     (richardson-sequence (make-zeno-sequence f start-h)
                          ord
                          inc)
     tolerance))
```

We are ready now to apply our tools to a significant example.

Numerical computation of derivatives

The following higher-order procedure takes a procedure that computes a numerical function and returns a procedure that calculates an approximation to the derivative of that function:

```
(define (make-derivative-function f)
  (lambda (x)
    (let ((h .00001))
      (/ (- (f (+ x h)) (f (- x h)))
         2 h))))
```

Notice the ad hoc definition of `h`. We are walking the line between truncation error (not having `h` small enough for the difference quotient to adequately approximate the derivative) and roundoff error (having `h` so small that the subtraction of nearly equal quantities loses all accuracy in the answer). The optimal `h` depends both on `x` and on the number of digits carried by the machine, but even with this `h` we'll generally lose about a third of our significant digits (we'd lose half of our digits had we used the forward, rather than the centered difference quotient). Of course we're hoping that Richardson will allow us to do better.

It's instructive to experiment with letting `h` go to 0. Given `f`, `x`, and `h`, we produce a stream of difference quotients in which `h` is successively reduced by a factor of 2.

```
(define (diff-quot-stream f x h)
  (cons-stream (/ (- (f (+ x h)) (f (- x h))) 2 h)
               (diff-quot-stream f x (/ h 2))))
```

We apply this to the estimation of the derivative of the square root at 1 (exact answer is 0.5).

```
(print-stream (diff-quot-stream sqrt 1 .1))

  ==>   0.500627750598189
        0.500156421150633
        0.500039073185090
        0.500009766292631
        0.500002441447984
        0.500000610354192
        0.500000152587994
        0.500000038146951
        0.500000009536734
        0.500000002384411
        0.500000000595833
        0.500000000148475
        0.500000000038199
        0.500000000010914       ; 14th term
        0.500000000010914
        0.499999999974534
        0.499999999992724
        0.500000000029104
        0.499999999883585
        ...
        0.500183105468750       ; 40th term
        0.500488281250000
        0.499267578125000
        0.498046875000000
        0.502929687500000
        0.507812500000000
        0.488281250000000
        0.468750000000000
        0.546875000000000
        0.625000000000000
        0.312500000000000       ; 50th term
        0.
        0.
        0.
        ...
```

We observe that the error diminishes steadily until the 14th term is reached; after this, the error builds back up in a somewhat erratic man-

ner until, after the 50th term, we are left with a steady parade of zeros. This problem results from the subtraction of nearly equal quantities in the numerator of the difference quotient: we lose more and more significant figures until h becomes so small that $x + h$ and $x - h$ are equal to full working precision, after which only 0 quotients can be returned.

Hence we are in a race between truncation error, which starts out large and gets smaller, and roundoff error, which starts small and gets larger. Richardson helps the situation by creating new sequences in which the truncation error dimishes more rapidly, which is just what we need. To be more precise, we need to look at how the roundoff error works in this example.

Any real number x, represented in the machine, is rounded to a value $x(1 + e)$, where e is effectively a random variable whose absolute value is on the order of the *machine epsilon*, ϵ: that smallest positive number for which 1.0 and $1.0 + \epsilon$ can be distinguished. For IEEE double precision (as used, for example, by the 8087 numeric coprocessor), $\epsilon = 2^{-53} = 1.11 \times 10^{-16}$. Now if h is small, both $f(x + h)$ and $f(x - h)$ have machine representations in error by around $f(x)\epsilon$; their difference suffers an absolute error of this same order. Since the difference $f(x + h) - f(x - h)$ *should* equal around $f'(x)2h$, the relative error is of the order

$$\epsilon \left| \frac{f(x)}{2hf'(x)} \right|$$

The relative error of the difference quotient is essentially the same as that of its numerator, the denominator being just $2h$ which is known to full precision.

From the above expression, we see that the relative error due to roundoff basically doubles each time h is halved—a result that is easy to see directly in terms of the binary representation of $x + h$: dividing h by 2 shifts the binary representation of h one position to the right; but the presence of x nails down the high order bits of $x + h$, whence the low order bits of h fall off the end, one per iteration.

Suppose we want to compute the derivative of the square root at 1 with a relative error of at most 10^{-13}, and starting with $h = 0.1$. We need to estimate the initial relative roundoff error; the preceding formula must be modified slightly for this purpose. First, the denominator is actually $f(x + h) - f(x - h)$, which is what we must use (it was written above as $2hf'(x)$ only to show the trend as h gets small). Second, we want to ensure that

the predicted relative roundoff error is at least a positive multiple of the machine epsilon; hence we take the *next-highest-integer* of the absolute-value subexpression:

$$1 + floor(\left|\frac{\sqrt{1}}{\sqrt{1.1} - \sqrt{0.9}}\right|) = 10$$

Thus, the initial relative roundoff error is 10 *roundoff units*, or $10\epsilon = 1.1 \times 10^{-15}$. Since the roundoff error roughly doubles at each iteration, we ask: How many times can 1.1×10^{-15} be doubled before reaching 10^{-13}?

$$1.1 \times 10^{-15} 2^n \leq 10^{-13}$$

so

$$n \leq \frac{\log(10^{-13}/1.1 \times 10^{-15})}{\log 2} = 6.5$$

Hence if we restrict ourselves to at most 6 terms past the first (for a total of 7 terms), we can be reasonably sure our data is uncontaminated by noise at the level of interest. This makes accelerated convergence really crucial: we have to reach our limit quickly or not at all.

Here's a modified version of `stream-limit` that accepts an optional final parameter `m`, designating the maximum number of stream terms to examine in the search for the limit. If `m` is reached without convergence, we just return the final term as a best guess; a more professional approach would be to return some kind of an error code, along with the best guess and an estimate of its truncation error.

```
(define (stream-limit s tolerance . opts) ;opts = optional args
  (let ((M (if (null? opts) 'nomax (car opts))))
    (let loop ((s s) (count 2))
      (let* ((h1 (head s)) (t (tail s)) (h2 (head t)))
        (if (close-enuf? h1 h2 tolerance)
            h2
            (if (and (number? M) (>= count M))
                h2
                (loop t (+ count 1))))))))
```

The revised version of `richardson-limit` is simply:

```
(define (richardson-limit f start-h ord inc tolerance . opts)
  (stream-limit
     (richardson-sequence (make-zeno-sequence f start-h)
                          ord
                          inc)
     tolerance
     (if (null? opts) 'nomax (car opts))))
```

We can now define our derivative estimator in the following natural way. (In practice, the routine as shown admits of several pitfalls: h becomes 0 if x is; `delta` might end up as 0 by chance; and possibly some other bad things we haven't thought of. It will serve for purposes of illustration here.)

```
(define rderiv
  (lambda (f tolerance)
    (lambda (x)
      (let* ((h (* 0.1 (abs x)))
             (delta (- (f (+ x h)) (f (- x h))))
             (roundoff (* *machine-epsilon*
                          (+ 1 (floor (abs (/ (f x) delta))))))
             (n (floor (/ (log (/ tolerance roundoff))
                          (log 2)))))
        (richardson-limit (lambda (dx)
                            (/ (- (f (+ x dx))
                                  (f (- x dx)))
                               (* 2 dx)))
                          h
                          2
                          2
                          tolerance
                          (+ n 1))))))
```

Notice that the `ord` and `inc` arguments are both 2: the truncation error involves only even powers of h. Had we used the forward difference quotient, $(f(x+h)-f(x))/h$, then all powers of h would arise, and `ord` and `inc` would both be 1. These results follow easily from the Taylor expansion of f.

Applied to the square root example, we find:

```
((rderiv sqrt 1e-13) 1) ==> 0.500000000000016
```

which shows a relative error of 0.32×10^{-13}. Further testing shows that the relative error of 10^{-13} is generally met.

We pass now to another significant application, in which roundoff error is happily not an issue.

Numerical integration by Romberg's method

Given a function f that behaves nicely (*i.e.*, has two continuous derivatives) on a finite interval $[a, b]$, we seek numerical approximations to the definite integral of f from a to b. The plan of attack is to divide $[a, b]$ into some number, n, of subintervals each of length $h = (b - a)/n$. We apply the trapezoidal rule to compute an approximating sum S_n; we then form a sequence of approximations by repeatedly doubling n (equivalently, halving h) and use Richardson extrapolation on the result.

We get the ball rolling with the following procedure. It takes f, a, and b, and returns a procedure that, given n, computes S_n:

```
(define (trapezoid f a b)
  (lambda (n)
    (let ((h (/ (- b a) n)))
      (let loop ((i 1) (sum (/ (+ (f a) (f b)) 2)))
        (let ((x (+ a (* i h))))
          (if (< i n)
              (loop (+ i 1) (+ sum (f x)))
              (* sum h)))))))
```

We use this to estimate π:

```
(define (f x) (/ 4 (+ 1 (* x x))))
(define pi-estimator (trapezoid f 0 1))

(pi-estimator 10)    ==> 3.13992598890716
(pi-estimator 10000) ==> 3.14159265192314
```

It is shown in standard texts (for example [2] or [3]) that, for f in $C^2[a, b]$ as we've assumed, the truncation error involves only even powers of h. Hence we proceed:

```
(define (pi-estimator-sequence n)
  (cons-stream (pi-estimator n)
               (pi-estimator-sequence (* 2 n))))
```

```
(print-stream
  (richardson-sequence
    (pi-estimator-sequence 10) 2 2))

  ==>     3.13992598890716
          3.14159265296979
          3.14159265362079
          3.14159265358979
          ...
```

The convergence rate is very encouraging—we get full machine accuracy in only 80 evaluations of the original function f—but there is considerable redundant computation here. Every time we double the number of points we reevaluate the integrand at the old grid points; this is a lamentable inefficiency in cases where f is expensive to compute. Romberg's method of quadrature, to which we now proceed, is a variation of the above that avoids unnecessary recomputation of f.

We begin with a utility procedure that computes a sum of terms $f(i)$, where i takes unit steps from a to (not-greater-than) b:

```
(define (sigma f a b)
  (let loop ((sum 0) (x a))
    (if (> x b)
        sum
        (loop (+ sum (f x)) (+ x 1)))))
```

We examine how the sum S_{2n} is related to S_n. In the former, we employ a partition of $[a, b]$ into $2n$ equal parts, having grid points at $x_i = a + ih$, with $h = (b - a)/2n$, and i going from 0 to $2n$. The even-index terms of this point sequence comprise the entire point sequence for S_n; in computing S_{2n}, we need only evaluate f at the odd indices, the others being already incorporated into S_n:

$$S_{2n} = \frac{1}{2}S_n + h\sum_{i=1}^{n} f(x_{2i-1})$$

This recursion is the basis of the following procedure, which generates the sequence $S_1, S_2, S_4, \cdots$:

```
(define (trapezoid-sums f a b)
  (define (next-S S n)
    (let* ((h (/ (- b a) 2 n))
           (fx (lambda(i) (f (+ a (* (+ i i -1) h))))))
      (+ (/ S 2) (* h (sigma fx 1 n)))))
  (define (S-and-n-stream S n)
    (cons-stream (list S n)
                 (S-and-n-stream (next-S S n) (* n 2))))
  (let* ((h (- b a))
         (S (* (/ h 2) (+ (f a) (f b)))))
    (map-stream car (S-and-n-stream S 1))))
```

We arrive at the more economical version of our previous method:

```
(define (romberg f a b tolerance)
  (stream-limit
    (richardson-sequence (trapezoid-sums f a b)
                         2
                         2)
    tolerance))
```

Conclusion

We have shown how a classical numerical analysis method, Richardson extrapolation, can be formulated as a package of procedures that can be used as interchangable components in the construction of traditional applications such as the estimation of a derivative and Romberg quadrature. Such a formulation is valuable in that it separates out the ideas into several independent pieces, allowing one to mix and match combinations of components in a flexible way to facilitate attacking new problems. Clever ideas, such as Richardson extrapolation, need be coded and debugged only once, in a context independent of the particular application, thus enhancing the reliability of software built in this way. Roylance [5] has similar goals. He constructs high-performance implementations of special functions, abstracting out recurrent themes such as Chebyshev economization.

The decompositions we displayed have used powerful abstraction mechanisms built on high-order procedures, interconnected with interfaces organized around streams. The programs were implemented functionally—there

were no assignments or other side-effects in any of our example programs. Because functional programs have no side effects they have no required order of execution. This makes it exceptionally easy to execute them in parallel.

A program is a communication, not just between programmers and computers, but also between programmers and human readers of the program; quite often, between the programmer and him/herself. A program describes, more or less clearly, an idea for how to obtain some desired results. One power of programs is that they allow one to make the knowledge of methods explicit, so that methods can be studied as theoretical entities. Traditional numerical programs are hand crafted for each application. The traditional style does not admit such explicit decomposition and naming of methods, thus losing a great part of the power and joy of programming.

Appendix: Convergence rate of the first-terms sequence

We offer here the mathematical justification for claims made earlier about the rate of convergence of the sequence of first terms returned by the procedure `richardson-sequence`.

As for hypotheses, we suppose that R is a function analytic in a disk that contains h as an interior point; thus we have an expansion

$$R(h) = A + \sum_{i=1}^{\infty} E_i h^{p_i},$$

where $\{p_1, p_2, \cdots\}$ is an increasing sequence of positive integers. We assume the p_i are chosen so that no E_i is 0. Let us identify R with $R^{[1]}$; the sequence of first terms is given by

$$S = \{R^{[1]}(h), R^{[2]}(h), \cdots, R^{[n]}(h), \cdots\}$$

where for all $n > 0$,

$$R^{[n+1]}(h) = \frac{2^{p_n} R^{[n]}(\frac{h}{2}) - R^{[n]}(h)}{2^{p_n} - 1}.$$

This is the operation concocted to remove the dominant error term at each stage; thus we know

$$S_n = R^{[n]}(h) = A + \sum_{i=n}^{\infty} E_i^{[n]} h^{p_i}.$$

Since it is possible that the coefficients $E_i^{[n]}$ grow large as $n \to \infty$, we cannot immediately conclude that S_n converges to A or even converges at all. We settle this question by a straight-forward computation.

To begin with, we have

$$S_2 = A + \frac{1}{2^{p_1} - 1} \sum_{i=2}^{\infty} E_i \left(\frac{1}{2^{p_i - p_1}} - 1\right) h^{p_i}$$

and

$$S_3 = A + \frac{1}{(2^{p_1} - 1)(2^{p_2} - 1)} \sum_{i=3}^{\infty} E_i \left(\frac{1}{2^{p_i - p_1}} - 1\right)\left(\frac{1}{2^{p_i - p_2}} - 1\right) h^{p_i};$$

the general case is seen to be

$$S_{n+1} = A + \left(\prod_{i=1}^{n} \frac{1}{2^{p_i} - 1}\right) \sum_{i=n+1}^{\infty} E_i \left\{\prod_{j=1}^{n} \left(\frac{1}{2^{p_i - p_j}} - 1\right)\right\} h^{p_i}.$$

The term appearing in braces is less than 1 in magnitude; this gives us the estimate

$$|S_{n+1} - A| \leq \left(\prod_{i=1}^{n} \frac{1}{2^{p_i} - 1}\right) \sum_{i=n+1}^{\infty} |E_i h^{p_i}|.$$

Since R is analytic at h, the summation part

$$\mu_n = \sum_{i=n+1}^{\infty} |E_i h^{p_i}|$$

converges monotonically to 0 as $n \to \infty$. The product appearing in parentheses is estimated as follows.

Let $\sigma_n = \sum_{i=1}^{n} p_i$; then

$$\prod_{i=1}^{n} \frac{1}{2^{p_i} - 1} = \frac{1}{2^{\sigma_n}} \prod_{i=1}^{n} \left(\frac{1}{1 - 2^{-p_i}}\right).$$

Using the inequality, valid for $x \in [0, \frac{1}{2}]$,

$$\frac{1}{1 - x} \leq 1 + 2x,$$

we see that the product above is dominated, in magnitude, by

$$\frac{1}{2^{\sigma_n}}\prod_{i=1}^{n}(1+\frac{2}{2^{p_i}}) < \frac{1}{2^{\sigma_n}}\prod_{i=1}^{\infty}(1+\frac{2}{2^{p_i}}) \leq \frac{1}{2^{\sigma_n}}\prod_{i=0}^{\infty}(1+\frac{1}{2^i}) = \frac{K}{2^{\sigma_n}}$$

where $K = \prod_{i=0}^{\infty}(1+2^{-i})$ is an absolute constant.

Thus we have shown that the absolute error with which S_{n+1} approximates the limit A is less than

$$\frac{K\mu_n}{2^{\sigma_n}}.$$

In the cases cited in our discussion, $\{p_i\}$ is an arithmetic progression $\{p + (i-1)q\}$; hence

$$\sigma_n = \sum_{i=1}^{n}(p+(i-1)q) = np + \frac{n(n-1)}{2}q.$$

This justifies our earlier claim.

References

[1] H. Abelson and G.J. Sussman, with J. Sussman. *Structure and Interpretation of Computer Programs.*, MIT Press, Cambridge MA, 1985.

[2] G. Dahlquist and A. Bjork. *Numerical Methods.*, Prentice-Hall, 1974.

[3] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1986.

[4] J. Rees and W. Clinger, et. al. *Revised^3 Report on the Algorithmic Language Scheme.* ACM SIGPLAN Notices, Vol. 21, no. 12. Dec 1986, pp. 37-79. also MIT AI Memo 848a, September 1986.

[5] G. L. Roylance. *Expressing Mathematical Subroutines Constructively.* MIT AI Memo 999, November 1987.

END
DATE
9-88
DTIC